Le DevOps à la rescousse de votre dette technique.

Agile Tour Québec 2019

Vincent Vial

Dette ou pas dette ?

Qu’est-ce que la dette technique ?

Métaphore inventée par **Ward Cunningham.**
Choix technologique qui est susceptible d’empêcher l’atteinte ou de retarder l’atteinte d’un objectif futur.

- ➔ Intentionnelle : « code jetable », « il faut livrer à cette date absolument», « on y reviendra plus tard », « on skip cet upgrade », non respect de bonne pratiques.
- ➔ Non intentionnelle : méconnaissance des bonnes pratiques, changement extrinsèque (fin de support d’une dépendance, …)

Qu’est-ce que n’est pas la dette technique ?

- Pas une question de technologie : mais de maîtrise de la technologie.
- Dette technique != Mauvais code : la dette technique ne se limite pas au code (absence d’automatisation, de données de tests exploitables, …).
- La dette technique n’est pas uniformément répartie dans une application.
- Se justifie parfois (*cf article de Vincent Driessen : « Technical debt is real debt »*) .

Le cercle vicieux de la dette technique...

Préoccupation technique ignorée

Les fonctionnalités sont développées plus lentement

Attrition des talents

Développer devient pénible et répétitif

Zombification du produit

** illustration tirée de la série The walking dead*

Quelles conséquences ?

Les conséquences de la dette technique

La Faillite technique (l'impossibilité de payer les intérêts de la dette technique) après une longue « Agonie Technique », il n'est plus possible de livrer les fonctionnalités nécessaires pour maintenir un produit viable.

- ➔ Rachat par la concurrence
- ➔ Rachat du logiciel par ses utilisateurs
- ➔ Refonte
- ➔ Achat d'un progiciel

Que faire pour résoudre ses problèmes de dettes ?

S’attaquer à la dette technique – Dimension culturelle

- favoriser la collaboration (Dev Talk, Pair Programming) : le développement c’est une activité créative réalisée à plusieurs.
- ne pas chercher à être parfait
- mesurer l’expérience acquise et le savoir acquis autant que le résultat
- Attention au syndrome de la vitre brisée

Most programmers want to write good code; but believe that their employers don't want good code. They are wrong. Most employers want the benefits of good code; but don't know that good code provides those benefits.

Uncle Bob Martin – 2019/06/25

S'attaquer à la dette technique – Volet technique

- Ne pas contracter plus de dette : implanter des essais automatisés de grande portée puis de portée plus petite.

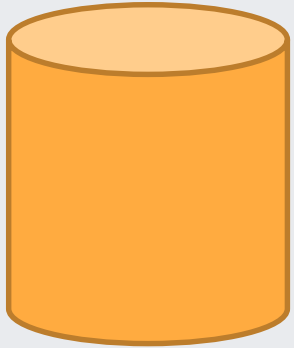
- Le « refactoring » est le remboursement de la dette :
  - opportuniste
  - proportionné
  - discipline
  - ne pas sous estimer l'impact parfois négatif du mot

- Investir dans le DevOps pour réduire la boucle de rétroaction sur son développement.

Et le DevOps dans tout ca ?

Hégémonie des base de données dans les systèmes « Legacy »

- ➔ utilisée par plusieurs applications
  avec parfois des DAL différentes
- ➔ contient de la logique d'affaire
- ➔ responsabilité partagée entre plusieurs équipes
  (collaboration difficile, instabilité des environnements)
- ➔ communication BD à BD .....
- ➔ Paradoxalement, la BD est le parent pauvre des tests et des processus de CI-CD

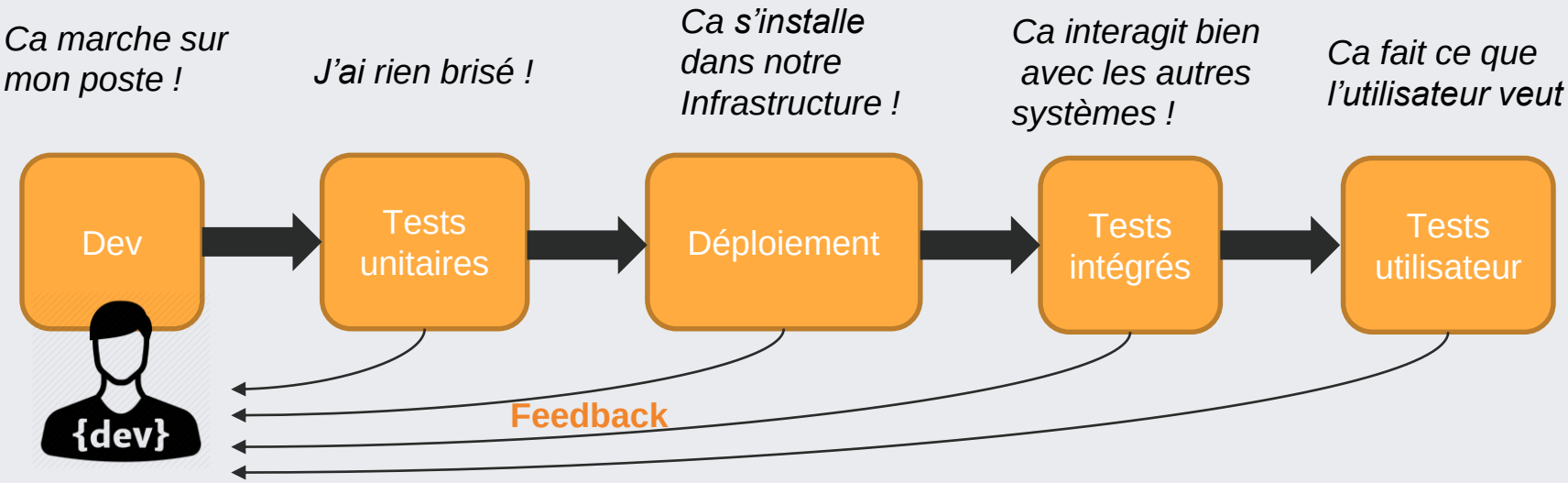


Interdépendance dans les systèmes « Legacy »

- ➔ Beaucoup d'interdépendances
- ➔ absence d'urbanisation (SOA, ...)
- ➔ collaboration entre les équipes difficiles du fait des environnements partagés
- ➔ Les tests d'intégration nécessitent la cohérence entre les jeux de données entre plusieurs systèmes.

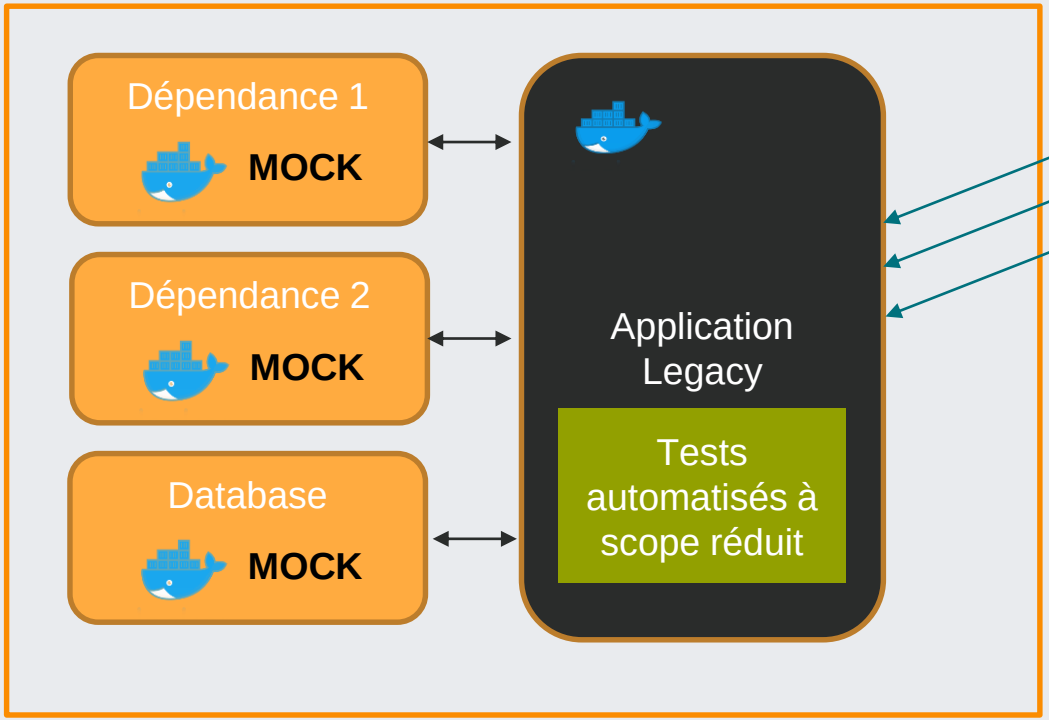
La démarche itérative et incrémentale nécessaire au « refactoring » pour réduire la dette technique est alors hasardeuse.

« Feedback loop » en développement

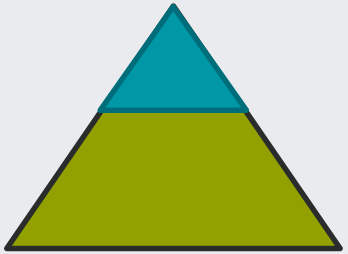


La rapidité avec laquelle le développeur reçoit le « **feedback** » est directement reliée à sa **productivité.**

Méthode de « refactoring » d'une application « Legacy »



Tests de caractérisation automatisés à large scope (filet de sécurité)



DevOps et Dette Technique

- ➔ En réduisant le délais pour le développeur pour avoir un **feedback** sur son travail, il peut régler plus de dette à chaque itération.
- ➔ En rapprochant son environnement **local** de la réalité d'un environnement de **production**, le développeur atteint le résultat en moins d'itérations.
- ➔ En facilitant la création d'environnement permettant de tester, les environnements partagés avec les autres équipes sont plus stables.

Non seulement le DevOps permet de régler plus de dette dans le code mais l'absence de DevOps empêche de régler la dette technique.

Démos.

Démos avec Azure DevOps

Démo 1

« Evolutionary Dabase Design »

Démo 2

« Mocker » une dépendance.

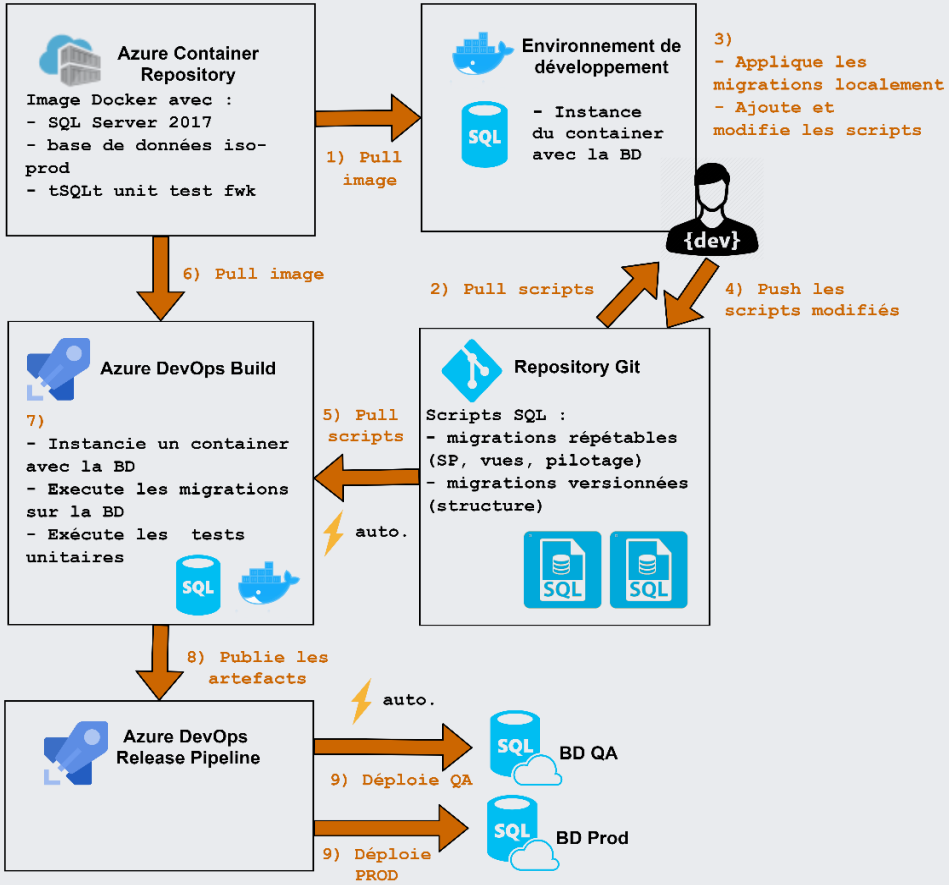
Démo 3

« Scaler » son serveur de build.

Démo 1 : « Evolutionary Database Design »

- Principe évoqué par Martin Fowler dans l'article : https://martinfowler.com/articles/evodb.html
- Une base de donnée (au moins) par développeur, pouvant être mis à jour facilement
- Intégrer les changements en continue
- Versionner tous les éléments constitutifs de la base de données

Démo 1 : « Evolutionary Database Design »



Démo 2 : « Mocker » une dépendance